TD 7

Diviser pour régner

Correction

Exercice 1 :

On considère un anneau $\mathcal{A}$. Pour tout élément e de cet anneau, pour tout entier n , on note e^n l'élément de $\mathcal{A}$ valant $\underbrace{eee...e}_{n \text{ fois}}$.

On veut écrire une fonction puis permettant d'élever un élément de l'anneau à la puissance n. Cette fonction prendra comme argument :

- Un élément e, élément de l'anneau, et un entier n ;
- Une fonction `fois`, qui permet d'effectuer la multiplication de l'anneau.
- Un élément `id`, qui sera l'élément neutre de la multiplication de l'anneau.

La fonction `puis` sera donc du type :

```
'a -> int -> ('a->'a->'a)->'a->'a
```

1. Écrire la fonction `puis` en utilisant l'algorithme naïf. Combien de fois utilise-t-on la fonction `fois` ?

   Correction

   L'algorithme naïf consiste à calculer la puissance en faisant autant de "fois" que nécessaire.

   Listing 1 – La fonction `puis`

```
let puis e  n fois id =
  let p = ref id in
  for i = 1 to n do
    p := fois !p e
  done ;
  !p;;
```

   Donc il y a n utilisation de la fonction `fois`.

2. On considère l'algorithme suivant :

   - Si $n = 0$ alors `puis e 0 fois id` renvoie `id` ;
   - Si $n = 1$ alors `puis e 1 fois id` renvoie `e` ;
   - Si $n = 2k$, on pose `e'` $=$ `puis e k fois id` on calcule alors `fois e' e'` ;
   - Si $n = 2k+1$, on pose `e'` $=$ `puis e k fois id` on calcule alors `fois e fois e' e'` ;

   (attention, il est important de poser e' pour ne pas faire deux fois un calcul identique !)

   (a) Combien de fois la fonction `fois` sera-t-elle utilisée si on calcule e^{10} ? Et pour e^{15} ? Et pour e^{16} ?

   Correction

   - pour e^{10} on compte 4 fois.

- pour e^{15} on compte 6 fois.
- pour e^{16} on compte 4 fois.

(b) Montrer la terminaison et la correction de l'algorithme.

Correction

Pour $n \in \mathbb{N}$, on pose $P_n =$ " puis e n se termine et renvoie e^n ".

- On a évidement P_0 et P_1.
- On suppose que la propriété est vraie pour tout $k < n$. Montrons que la propriété reste vraie pour n :
  - Si n est paire, on a donc $n = 2k$. Par hypothèse, on sait que P_k est vraie. On a `puis e n` $=$ `fois e' e'` , avec `e'` $=$ `puis e k`, le calcul de `e'` se termine et renvoie e^k, donc `puis e n` se terminer et renvoie $e^k \times e^k = e^{2k} = e^n$. Donc P_n est vraie.
  - Si n est impaire, on a donc $n = 2k+1$. Par hypothèse, on sait que P_k est vraie. On a `puis e n` $=$ `fois e fois e' e'` , avec `e'` $=$ `puis e k`, le calcul de `e'` se termine et renvoie e^k, donc `puis e n` se terminer et renvoie $e \times e^k \times e^k = e^{2k+1} = e^n$. Donc P_n est vraie.

  Donc la propriété est vrais pour n.

On conclue donc, par récurrence, que, pour tout n la fonction se termine et renvoie e^n.

(c) On note c_n le nombre de `fois` utilisé pour le calcul de e^n. On note $n = \overline{b_p b_{p-1} ... b_0}^2$ la décomposition de n en base 2.

i. Déterminer c_0 et c_1.

ii. Montrer que l'on a, pour $n > 1$:

$$c_n = c_{\lfloor n/2 \rfloor} + 1 + b_0$$

Correction

Pour $n = \overline{b_p b_{p-1} ... b_0}^2$, on a naturellement $\lfloor n/2 \rfloor = \overline{b_p b_{p-1} ... b_1}^2$.
On a donc :

- si n est pair :$c_n = c_{\lfloor n/2 \rfloor} + 1$, car on pose alors $n = 2k$, avec `e'` $=$ `puis e k` qui compte $c_{\lfloor n/2 \rfloor}$ `fois` , on ajoute un `fois` dans `fois e' e'`. De plus, n étant pair, $b_0 = 0$, on retrouve donc bien $c_n = c_{\lfloor n/2 \rfloor} + 1 + b_0$.
- si n est impair : $c_n = c_{\lfloor n/2 \rfloor} + 2$, car on pose alors $n = 2k+1$, avec `e'` $=$ `puis e k` qui compte $c_{\lfloor n/2 \rfloor}$ `fois` , on ajoute deux `fois` dans `fois e fois e' e'`. De plus, n étant impair, $b_0 = 1$, on retrouve donc bien $c_n = c_{\lfloor n/2 \rfloor} + 1 + b_0$.

iii. En déduire que, pour tout $n \in \mathbb{N}$:

$$c_n = p + \sum_{k=0}^{p-1} b_k$$

Correction

Pour $n = \overline{b_p b_{p-1} ... b_0}^2$, on a naturellement $\lfloor n/2 \rfloor = \overline{b_p b_{p-1} ... b_1}^2$.
On a donc :

- si n est pair :$c_n = c_{\lfloor n/2 \rfloor} + 1$, car on pose alors $n = 2k$, avec `e'` $=$ `puis e k` qui compte $c_{\lfloor n/2 \rfloor}$ `fois` , on ajoute un `fois` dans `fois e' e'`. De plus, n étant pair, $b_0 = 0$, on retrouve donc bien $c_n = c_{\lfloor n/2 \rfloor} + 1 + b_0$.

- si n est impair : $c_n = c_{\lfloor n/2 \rfloor} + 2$, car on pose alors $n = 2k+1$, avec $e' =$ `puis e k` qui compte $c_{\lfloor n/2 \rfloor}$ `fois` , on ajoute deux `fois` dans `fois e fois e' e'`. De plus, n étant impair, $b_0 = 1$, on retrouve donc bien $c_n = c_{\lfloor n/2 \rfloor} + 1 + b_0$.

(d) Montrer que le nombre d'utilisation de la fonction `fois` dans le calcul de e^n est inférieur à $2\lfloor \log_2(n) \rfloor$.

Correction

Pour $n = \overline{b_p b_{p-1} ... b_0}^2$,on remarque que $p = \lfloor \log_2(n) \rfloor$ (en effet, on a $2^p \leq n < 2^{p+1}$, donc par croissance du logarithme, on a $\log_2(2^p) \leq \log_2(n) < \log_2(2^{p+1})$ ce qui donne bien $p \leq \log_2(n) < p+1$).

De plus, $\forall k \in [\![0, p-1]\!]$, $b_k \leq 1$, donc $\sum_{k=0}^{p-1} b_k \leq p$, soit $\sum_{k=0}^{p-1} b_k \leq \lfloor \log_2(n) \rfloor$.

On a donc bien, $c_n \leq \lfloor \log_2(n) \rfloor$.

(e) Écrire la fonction `puis`

Correction

Listing 2 – La fonction `puis`

```
let rec puis e n fois id = match n with
  | 0 -> id
  | 1 -> e
  | n when n mod 2=0 -> let e' = puis e (n/2) fois id in fois e' e'
  | _-> let e' = puis e ((n-1)/2) fois id in fois e (fois e' e')
;;
```

Exercice 2 :

Dans le même esprit on retrouve la recherche par dichotomie d'un entier. Écrire la fonction `recherche`, qui recherche la valeur d'un entier pris aléatoirement sur l'intervalle $[0, 100]$ et renvoie le nombre d'appel récursif ainsi que le nombre à rechercher.

```
recherche : unit -> int * int
```

Exemple :

```
recherche ();;
- : int * int = 7, 55
```

Exercice 3 :

On représente un polynôme sous la forme de la liste de ses coefficients, la tête correspondant au degré zéro :

```
type poly = float list;;
```

Ainsi, la liste `3 :: 4 :: 5 :: []` correspond au polynôme $3 + 4X + 5X^2$.

1. Opérations préliminaires :
   Écrire des fonctions récursives suivantes :

`normalise : poly->poly`	Une fonction qui retire les zéros inutiles en fin de liste.
`ajout : poly->poly`	Somme polynômiale
`mulscal:float->poly->poly`	Multiplication d'un polynôme par un scalaire
`soustr:poly->poly->poly`	Différence de deux polynômes.

2. Multiplication naïve :
   La multiplication de deux polynômes $P = \sum_k a_k X^k$ et $Q = \sum_k b_k X^k$ est donnée par la formule :
$$PQ = \sum_k \sum_{i+j=k} a_i b_j X^k$$
   Mais la représentation des polynômes sous forme de liste rend cette formule rédhibitoire (car le simple accès à un coefficient quelconque du polynôme se fait en temps linéaire). Cependant, on sait accéder en temps constant au coefficient de degré zéro. On peut donc écrire :$P = P_1 X + a_0$ et $Q = Q_1 X + b_0$ avec P_1 et P_2 deux polynômes.
   À l'aide du développement de $(P_1 X + a_0)(Q_1 X + b_0)$, écrire une fonction récursive :

   ```
   mult : polynome -> polynome -> polynome
   ```

   qui renvoie le produit de deux polynômes.

3. Algorithme de Karatsuba (1960) :
   On s'intéresse ici à une autre façon de décomposer un polynôme.
   Pour tout entier k, si on a $P = RX^k + S$ et $Q = TX^k + U$,alors le produit s'écrit :
$$PQ = X^{2k}RT + X^k(RU + ST) + SU$$
   En négligeant la multiplication par un X^j , cette méthode naïve requiert quatre multiplications de polynômes plus petits (si on impose les degrés de S et U strictement inférieurs à k). Or, Karatsuba (1960) a remarqué que, si on écrit ce produit sous la forme : $PQ = X^{2k}RT + X^k((R+S)(T+U)) + SU$ alors le produit ne requiert que

trois multiplications (au lieu de quatre naïvement) de polynômes plus petits : RT, SU et $(R+S)(T+U)$.
Cette remarque ouvre la voie à une implémentation en diviser pour régner.

- Écrire la fonction `separe l k` qui renvoie le couple (`l1`, `l2`) où `l1` est la liste des k premiers termes, et `l2` le reste.

  ```
  separe : 'a list -> int -> 'a list * 'a list
  ```

  exemple :

  ```
  separe [1;2;3;4;5;6;7;8;9] 5;;
  - : int list * int list = [1; 2; 3; 4; 5], [6; 7; 8; 9]
  ```

- Écrire une fonction `produit_xj` tel que `produit_xj i p` renvoie le polynôme X^iP.
- Écrire une fonction récursive :

  ```
  karatsuba : int -> polynome -> polynome -> polynome
  ```

  telle que `karatsuba k p q` renvoie le produit de deux polynômes suivant une méthode diviser pour régner avec le paramètre k pour la décomposition de deux polynômes. On pourra supposer que k est une puissance de 2.

Exercice 4 : Tri fusion

Le tri repose sur le fait que pour fusionner deux listes (ou tableaux) triées dont la somme des longueurs est n, $n-1$ comparaisons au maximum sont nécessaires.
L'algorithme peut s'effectuer récursivement :

1. On découpe en deux parties à peu près égales les données à trier.
2. On trie les données de chaque partie.
3. On fusionne les deux parties.

La récursivité s'arrête à un moment car on finit par arriver à des listes de 1 élément et alors le tri est trivial.
Exemple de fusion : Fusionner [1 ;2 ;5] et [3 ;4] : On sait que le premier élément de la liste fusionnée sera le premier élément d'une des deux listes d'entrée (soit 1, soit 3), propriété non remarquable sur des listes non triées.
On compare donc 1 et 3 : 1 est plus petit.
[2 ;5] - [3 ;4] -> [1]
On compare 2 et 3 : 2 est plus petit.
[5] - [3 ;4] -> [1 ;2]
On compare 5 et 3 : 3 est plus petit.
[5] - [4] -> [1 ;2 ;3]
On compare 5 et 4 : 4 est plus petit.
[5] -> [1 ;2 ;3 ;4] puis [1 ;2 ;3 ;4 ;5]
Exemple simple, pour la liste [6 ;1 ;2 ;5 ;4 ;7 ;3] : À l'état initial on a les éléments un par un, on les fusionne deux à deux : ([6] [1]) ([2] [5]) ([4] [7]) [3] ; On obtient : ([1 ;6] [2 ;5]) ([4 ;7] [3]) que l'on fusionne deux à deux à nouveau et ainsi de suite : ([1 ;2 ;5 ;6] [3 ;4 ;7]) puis [1 ;2 ;3 ;4 ;5 ;6 ;7].

1. Écrire la fonction fusion de deux listes triées :

   ```
   fusion : 'a list -> 'a list -> 'a list
   ```

Quelle est le nombre d'itérations de la fonction ?

2. Écrire une fonction fendre qui renvoie pour une liste donnée deux listes de tailles identiques (à une valeur près)

   ```
   fendre : 'a list -> 'a list * 'a list
   ```

3. Calculer un ordre de grandeur du nombre d'itérations.

4. Écrire la fonction tri_fusion :

   ```
   tri_fusion : 'a list -> 'a list
   ```

   Estimer le nombre d'itérations nécessaire.